Time Complexity and Universal TMs

1 Our first complexity classes

Definition (Complexity class)**.** A *complexity class* is any collection of languages. Usually, though, we focus on complexity classes whose definition is of the form "all languages computable using at most such-and-such resources".

Definition (The TIME complexity class)**.** Let $t : \mathbb{N} \to \mathbb{R}_{\geq 0}$ be a function (e.g., $t(n) = n^2$ or $t(n) = n \log n$). We define the following complexity class:

$$\mathsf{TIME}(t(n)) = \{\text{languages } L : \exists \text{ a TM deciding } L \text{ in } O(t(n)) \text{ time}\}.$$

Example. Palindromes $\in \mathsf{TIME}(n^2)$.

Note. Notice that the $O(\cdot)$ is built into the definition. This is because constant factors are not too meaningful in time complexity (for example, you can generally halve the running time of any algorithm by squaring its tape alphabet size — think about it!). This convention also makes the notation neater.

Notation. The definition of $\mathsf{TIME}(t(n))$ depends quite significantly on our choice of 1-tape TMs as the official model for algorithms. Had we chosen, e.g., 2-tape TMs, then we would have had **Palindromes** $\in \mathsf{TIME}(n)$. Because of this strong model-dependence, we will not focus *too* much on very precise time classes like $\mathsf{TIME}(n^2)$; instead we will prefer ones that are insensitive to polynomial time changes, like "P" which we define next.

Let's now introduce what is perhaps the most important complexity class, P, which captures polynomial-time solvable decision problems:

Definition (The complexity class P)**.**

$$\mathsf{P} = \bigcup_{c \in \mathbb{N}} \mathsf{TIME}(n^c) = \text{all languages decidable in polynomial time by a Turing Machine.}$$

Happily, the complexity class P is very "robust" with respect to the choice of model for algorithms. By the simulations mentioned in Chapter (**??**), any language decidable in polynomial time by a multi-tape TM, or "C-like pseudocode", etc., is also solvable in polynomial time by a 1-tape TMs (and vice versa). Indeed, the Extended Church–Turing Thesis tells us this is also true for any "reasonable" model of computation. Thus (unlike with $\mathsf{TIME}(t(n))$), the set of languages in P does not change if you change your official computational model from 1-tape TMs to some other reasonable model.

Of course in the real world, and in the theoretical study of algorithms, we certainly care about the distinction between $O(n)$-time algorithms (fantastic!) and $O(n \log n)$-time algorithms (very good!) and $O(n^2)$-time algorithms (kind of okay) and $O(n^{10})$-time algorithms (almost useless in practice!). But in our study of complexity theory, we won't worry too much about these distinctions: the definition of P is very elegant, it serves as a reasonable robust notion of "time-efficient algorithms", and any language that is *not* even in this liberal class P is almost surely not solvable efficiently in practice.

In Chapter (**??**) we will study some of the *many* problems that *are* in P. But in this chapter we will begin to think about the question: what kind of problems are *not* in P?

2 Turing's Theorem, and a time-bounded variant

We should know at least one decision problem not in P: namely, **Halts**, the Halting Problem. It is not solvable by *any* algorithm, let alone a polynomial-time algorithm! What's more interesting, though, is to ask if there is some problem that *is* solvable by an algorithm in finite time, but is not solvable by any algorithm with polynomial running time. Indeed, there is; over the course of the next two chapters, we will come to prove the following:

Theorem (Special case of Time Hierarchy Theorem)**.** *There exists a language $L \subset \{0,1\}^*$ such that:*

- $L \in \mathsf{TIME}(3^n)$*;*
- $L \notin \mathsf{P}$*; indeed, L is not even in the class* $\mathsf{TIME}(1.1^n)$*.*

(To explain this "indeed': Note that if L were in P*, it would be in* $\mathsf{TIME}(n^c)$ *for some constant c; but then also L would be in* $\mathsf{TIME}(1.1^n)$*, because $O(n^c) \leq O(1.1^n)$ for any c.)*

Theorem (Special case of Time Hierarchy Theorem) is a special case of the *Time Hierarchy Theorem*, which we will prove in the next chapter. The conceptual content of the Time Hierarchy Theorem is:

More time lets you solve more languages.

Theorem (Special case of Time Hierarchy Theorem) just promises that a language $L \in \mathsf{TIME}(3^n) \setminus \mathsf{TIME}(1.1^n)$ *exists*, but in fact we will be able to show a reasonably natural decision problem that has this property. One such problem is — roughly speaking — "given as input a Turing Machine M and a string w, simulate $M(w)$ for 2^n steps". It should be reasonably intuitive that this task *can* be done in $O(3^n)$ time, but *cannot* be done in $O(1.1^n)$ time. However this task is not actually a decision problem/language, so we will have to work harder to prove what we want.

The idea of this the "impossibility" half of the Time Hierarchy Theorem is that there is no clever way of telling what a given piece of code M will do on a given input w; it seems you can only try simulating $M(w)$ to find out. This is the exact same idea as in Turing's proof of the unsolvability of the Halting Problem, and it will be a good warmup to first recall this proof before we get to its "time-bounded variant", the Time Hierarchy Theorem.

For recapping Turing's Theorem, let us make two small technical shifts. First, for simplicity we will restrict attention to simulating Turing Machines that use input alphabet $\{0,1\}$ and tape alphabet $\{0,1,\sqcup\}$. This is no big deal; as we know, larger input alphabets Σ can always be re-encoded using $\{0,1\}$, and Proposition (**??**) showed us that we can always reduce a large tape alphabet Γ to one that just has $\Gamma = \{0,1,\sqcup\}$. Second, instead of considering the Halting Problem, it will be slightly more convenient to consider the TM Acceptance Problem.

Definition (Standard-alphabet TM)**.** A *standard-alphabet* TM is one where the input alphabet is $\Sigma = \{0,1\}$ and the tape alphabet is $\Gamma = \{0,1,\sqcup\}$.

Definition (TM acceptance problem)**.** The *TM acceptance problem* is the language

$$\textbf{Accepts} = \{\langle M, w\rangle : M \text{ is a "standard-alphabet" TM, } w \in \{0,1\}^*, \text{ and } M(w) \text{ accepts}\}.$$

Remark. Here we are assuming we have a scheme for encoding Turing Machines by strings. This shouldn't be considered unusual; a Turing Machine is just another kind of mathematical object, and we assume we have encoding schemes for all mathematical objects. Morphett's source code format for TMs is a perfectly good way of encoding TMs by the ASCII alphabet. Very shortly we will see another reasonable encoding of standard-alphabet TMs.

A slight variant on Turing's Theorem on the Halting Problem shows that there is *no* algorithm that decides the language **Accepts**. Relatedly, at the end of the next chapter we will be able to show:

Theorem (**BoundedAccepts$_2$•** not in P)**.** *One language L that works in Theorem (Special case of Time Hierarchy Theorem) — i.e., a language that is in* $\mathsf{TIME}(3^n)$ *but not* $\mathsf{TIME}(1.1^n)$ *— is the following variant of **Accepts**:*

$$\textbf{\textit{BoundedAccepts}}_2\bullet = \{\langle M, w\rangle : M(w) \text{ accepts within } 2^{|w|} \text{ steps}\}.$$

3 Universal Turing Machines

Before getting to Turing's Theorem and the Time Hierarchy Theorem, it will be good to think about the "positive" aspect of Theorem (**BoundedAccepts$_2$•** not in P), the fact that **BoundedAccepts$_2$•** *can* be decided in $O(3^n)$ time. The basic idea here is that we can write reasonably efficient code whose job is to simulate a given Turing Machine. When this code is itself written in the TM programming language, it is called a *Universal Turing Machine*.

Definition (Universal Turing Machine)**.** A *universal Turing Machine $\mathcal{U}$* is a Turing Machine that takes as input $\langle M, w\rangle$, where M is a standard-alphabet TM and $w \in \{0,1\}^*$, and which simulates M running on w. Here "simulates" means that $\mathcal{U}(\langle M, w\rangle)$ loops if $M(w)$ loops, and otherwise comes to the same output (accept/reject state, or actual string output) as $M(w)$.

Remark. It's reasonable to instead require that a universal Turing Machine be able to simulate a TM using *any* input alphabet and tape alphabet, not just the "standard alphabets". But it will be slightly more convenient for us in this chapter to stick to standard-alphabet TMs.

In short, a universal TM is a TM interpreter, written in the TM programming language. Turing's original paper explicitly showed the following:

Theorem (Universal TM exists)**.** *(Turing, 1936.) There exists a universal Turing Machine.*

This theorem should not come as great surprise these days; we can certainly imagine writing a TM simulator in a high-level programming language, and then the Church–Turing Thesis assures us we could convert this high-level code to TM code. For the purposes of proving the Time Hierarchy Theorem, though, we will need to get into some of the low-level

details of universal Turing Machines. The reason is that we will need to know about the precise *time-efficiency* of simulating TM code on a TM.

Actually, most any way of designing a universal TM will produce one that's "reasonably efficient", by which we mean one where the running time of $\mathcal{U}(\langle M, w\rangle)$ is no more than $\text{poly}(|\langle M, w\rangle|)$ times the running time of $M(w)$. A good way to see this, and to understand the proof of Theorem (Universal TM exists), is to study an explicit universal Turing Machine. Luckily, the Morphett website includes a very nice one, written by David Bevan...

Exercise. Go to http://morphett.info/turing/turing.html and load the example program called "Universal Turing machine", by David Bevan. Study this example carefully, and also study its documentation.

The gist of how $\mathcal{U}(\langle M, w\rangle)$ operates is that at all times, $\mathcal{U}$ keeps on its tape a modified version of the *configuration* of $M(w)$. The modification is that not only does $\mathcal{U}$ insert the current state symbol into the tape contents just to the left of the head, $\mathcal{U}$ in fact inserts all of $\langle M\rangle$ (the encoding of M) just to the left of the head. Given this idea, it is moderately straightforward to see how $\mathcal{U}$ needs to operate, but do check out the documentation!

Notice that in the documentation for Bevan's universal TM $\mathcal{U}$, there is a careful description of the encoding scheme for TMs (with tape alphabet fixed to $\Gamma = \{\texttt{0}, \texttt{1}, \sqcup\}$). In particular, it is assumed that the states are simply numbered $1, 2, 3, \ldots, s$ for some s. Since the tape alphabet always has size 3, it means there are always exactly $3s$ "lines" of code. The encoding of these lines is fairly straightforward, but with one somewhat clever angle: the "new state" at the end of each line is not encoded by its number, but rather by its offset from the current state encoded (effectively) in unary.

As mentioned in the documentation, the encoding size of an s-state machine is $O(s^2)$ symbols. As is also mentioned, if $M(w)$ halts in t steps, then $\mathcal{U}$'s simulation of $M(w)$ takes time $O(s^3 \cdot t)$. In fact, with careful inspection, you can see that the time can also be bounded by $O(|\langle M\rangle|^2 \cdot t)$. We record this as a theorem below.

Remark. An *extremely* pedantic point: Our definition of "universal Turing Machine" specifies that the input is $\langle M, w\rangle$, meaning a string in $\{\texttt{0}, \texttt{1}\}^*$; however Bevan's $\mathcal{U}$ uses a TM encoding involving a dozen or so symbols. Of course, if we wanted, we could replace each of these symbols with some four-symbol encoding over $\{\texttt{0}, \texttt{1}\}$, and slightly rework Bevan's $\mathcal{U}$ to deal with this. In practice, this would make Bevan's code much more annoying to read and understand. In theory, however, let's pretend that we indeed do this. Then Bevan's $\mathcal{U}$ will itself be a standard-alphabet TM.

Theorem (Time-efficient universal Turing Machine)**.** *There exists a (standard-alphabet) universal Turing Machine $\mathcal{U}$ with the guarantee that if $M(w)$ halts in t steps, then $\mathcal{U}$'s simulation takes time $O(|\langle M\rangle|^2 \cdot t)$.*

Exercise. Think about extending the TM encoding format to allow for *any* size input and tape alphabets. It's reasonable to insist that the input alphabet symbols are numbered $0, 1, 2, \ldots, a$

and the tape alphabet symbols are numbered $0, 1, 2, \ldots, b$ (where $b > a$, with symbol b being the "blank" symbol).

4 Universal Turing Machines with a clock

Let's return to the "positive half" of Theorem (**BoundedAccepts$_2$•** not in P), which says that the language **BoundedAccepts$_2$•** *can* be solved in $O(3^n)$ time. Given the time-efficient universal Turing Machine described in Theorem (Time-efficient universal Turing Machine), you might think this is easy to prove, but actually there is still a catch. Suppose we try to decide whether $\langle M, w\rangle \in$ **BoundedAccepts$_2$•** just by running our universal TM $\mathcal{U}$ on $\langle M, w\rangle \ldots$

Proposition. *Suppose $\mathcal{U}$ is given as input $\langle M, w\rangle$, and write $n = |\langle M, w\rangle|$ for the length of this input. Suppose that $M(w)$* does *in fact accept within $2^{|w|}$ steps. Then $\mathcal{U}$ will correctly detect this after it itself runs for at most $O(n^2 \cdot 2^n) \leq O(3^n)$ steps.*

Proof. We use Theorem (Time-efficient universal Turing Machine). Since $n = |\langle M, w\rangle|$, we infer both $|\langle M\rangle| \leq n$ and $|w| \leq n$. Thus $M(w)$ takes at most $2^{|w|} \leq 2^n$ steps, and $\mathcal{U}(\langle M, w\rangle)$ takes at most $O(|\langle M\rangle|^2 \cdot 2^n) \leq O(n^2 \cdot 2^n) \leq O(3^n)$ steps. □

The trouble is, what if $M(w)$ does *not* accept within $2^{|w|}$ steps? If this is because $M(w)$ *rejects* within $2^{|w|}$ steps, we are still okay; $\mathcal{U}$ will detect this within $O(n^2 \cdot 2^n)$ steps. But what if $M(w)$ simply doesn't halt at all within $2^{|w|}$ steps? What if $M(w)$ happens to halt in $2^{2^{|w|}}$ steps? We don't want to blithely allow $\mathcal{U}$ to keep running for $2^{2^{|w|}}$ steps…

Important. To successfully decide **BoundedAccepts$_2$•** in $O(3^n)$ steps, we'll want to have $\mathcal{U}(\langle M, w\rangle)$ *cut off* its simulation after $2^{|w|}$ steps. That is, $\mathcal{U}$ will need to "time" itself, or "clock" itself.

Intuitively, it shouldn't be too hard to implement this extra feature into $\mathcal{U}$, but we *do* have to do a little work.

Theorem (Alarm-clocked universal Turing Machine)**.** *There exists an* alarm-clocked *universal Turing Machine $\mathcal{U}$ with the following properties: It takes as input $\langle t, M, w\rangle$, where the new input $t \in \mathbb{N}$ is a time-bound (encoded in base-2, as usual). $\mathcal{U}$ correctly simulates $M(w)$ up until $M(w)$ halts or until $M(w)$ has run for t steps, whichever comes first. Furthermore, $\mathcal{U}(\langle t, M, w\rangle)$ itself runs in at most $O(|\langle M\rangle|^2 + \log t) \cdot t$ steps.*

Proof. Recall that our basic universal Turing Machine $\mathcal{U}$ works by keeping the TM description $\langle M \rangle$ on the tape, in the position of M's tape head. The alarm-clocked variant will *also* keep a "countdown-timer" written in base-2, stored just to the left of $\langle M \rangle$ on the tape. Initially, this timer will be t. Recall that when $\mathcal{U}$ simulates one step of $M(w)$, it does $O(|\langle M \rangle|^2)$ steps of work walking around the TM description $\langle M \rangle$ to figure out exactly what $M(w)$ is doing on the step, and to update things appropriately. Now, in addition, $\mathcal{U}$ will decrement the countdown-timer by 1 on each step of $M(w)$. This takes an additional $O(\log t)$ time per step of $M(w)$, since the encoding length of the countdown-timer is at most $|\langle t \rangle| = O(\log t)$. Finally, if the countdown-timer ever reaches zero, $\mathcal{U}$ will know to halt the simulation. □

With an alarmed-clocked universal TM $\mathcal{U}$ in hand, we can now show that **BoundedAccepts$_{2\bullet}$** is solvable in $O(3^n)$ time. There is *still* a little work to be done, though: our code will need to *compute* the time-bound $2^{|w|}$ to give to $\mathcal{U}$.

Lemma (2^n is time-constructible)**.** *The function $f : \mathbb{N} \to \mathbb{N}$ defined by $f(m) = 2^m$ is computable by TM code C with the following property: on input $\langle m \rangle$, the code C takes $O(m \log m)$ steps.*

Important. In Lemma (2^n is time-constructible), contrary to what we almost always do, we did *not* describe C's running time as a function of its input *length*, $|\langle m \rangle|$ (which is approximately $\log_2 m$). Rather, we described it in terms of the numerical value m itself. Following our usual conventions, the running time function of C would be $T_C(n) = 2^n \cdot n$. Please make sure to understand this point thoroughly!

We will omit the proof of Lemma (2^n is time-constructible) here because it is virtually identical to Problem 1 on Homework 2. That problem basically asks for code which, given m, outputs m copies of a certain symbol in time $O(m \log m)$. If we choose that symbol to be "`0`", and then we stick a "`1`" on the left at the very end of the algorithm, we get an algorithm that produces the string $\texttt{1}\texttt{0}^m$, which is exactly the base-2 representation $\langle m \rangle$ of 2^m, as desired!

Theorem. ***BoundedAccepts$_{2\bullet}$*** $\in \mathsf{TIME}(3^n)$.

Proof. We describe TM code running in $O(3^n)$ time that correctly decides whether a given input $\langle M, w \rangle$ is in the language **BoundedAccepts$_{2\bullet}$**. Let n denote $|\langle M, w \rangle|$, so $|\langle M \rangle| \leq n$ and $|w| \leq n$. We describe the stages of our code:

Compute n stage. The first stage is to have the algorithm determine and write down $\ell := |w|$, in base-2. Note that the value of ℓ is at most n, and that the number of symbols required to write it down, $|\langle \ell \rangle|$, is $O(\log n)$. We omit further description of this stage, as it is very similar to Problem 4 on Homework 1; as you showed there, this stage can be done in time $O(n^2)$ (or time $O(n \log n)$ with some cleverness).

Compute $2^{|w|}$ stage. The next stage is to compute and write down the time-bound $t = 2^{|w|} = 2^\ell$, using Lemma (2^n is time-constructible) as a subroutine (with $m = \ell$). As that Lemma says, the time to do this is $O(\ell \log \ell) \leq O(n \log n)$, and the number of symbols needed to write down $2^{|w|}$ is basically $\log_2(2^{|w|}) = |w| = \ell \leq n$.

Simulation stage. Now that we have gotten $t = 2^{|w|}$ written down on the tape (in base 2), we can use the alarm-clocked universal TM $\mathcal{U}$ from Theorem (Alarm-clocked universal Turing Machine) with input $\langle t, M, w\rangle$. This indeed lets us correctly tell whether $M(w)$ accepts in at most $2^{|w|}$ steps (or else whether it rejects/doesn't halt). As Theorem (Alarm-clocked universal Turing Machine) says, this use of $\mathcal{U}$ takes time

$$O(|\langle M\rangle|^2 + \log t) \cdot t \leq O(n^2 + |w|) \cdot 2^{|w|} \leq O(n^2 + n) \cdot 2^n \leq O(n^2 \cdot 2^n) \leq O(3^n).$$

We see that the overall time for the three stages is $O(n^2) + O(n \log n) + O(3^n) \leq O(3^n)$. □

Important. Again, the issues surrounding the "time to compute the time bound $2^{|w|}$" can be a bit confusing, so please carefully study the proof above, always bearing in mind the distinction between the size of a number m, and the size of the binary string representing m (which is basically $\log_2 m$).

This gives us the "positive half" of Theorem (**BoundedAccepts$_{2\bullet}$** not in P). In the next chapter we will prove the "negative half", that **BoundedAccepts$_{2\bullet}$** $\notin$ TIME(1.1^n).

For the general Time Hierarchy Theorem, we would ideally like to prove an analogue of Theorem (**BoundedAccepts$_{2\bullet}$** not in P) for "any" two running time functions with a good gap between them, not just 3^n versus 1.1^n. Let's look again at our proof of Theorem (), the "positive half" that **BoundedAccepts$_{2\bullet}$** $\in$ TIME(3^n), and imagine trying to generalize. Specifically, suppose we consider the analogous language **BoundedAccepts$_{f(\bullet)}$** for some function $f(\bullet)$ other than $2^\bullet$.

Remark (Simulation stage running time)**.** Intuitively, the main work is the "Simulation stage". For our example $f(n) = 2^n$, this stage actually took $O(n^2 \cdot 2^n)$ time, which we somewhat sloppily upper-bounded by 3^n. For general $f(n)$, this main part will take at most $O(|\langle M\rangle|^2 + \log f(n)) \cdot f(n) \leq O(n^2 + \log f(n)) \cdot f(n)$ time, which you should think of as "a little bit more than $f(n)$ time". In the next two remarks, we'll look at whether the other two "initialization stages" will come in at less than $f(n)$ time.

Remark (Compute n stage running time)**.** The "Compute n stage" took $O(n^2)$ time. As mentioned on Problem 4 of Homework 1, you can get this down to $O(n \log n)$ time with some cleverness. But you can't do better than that, and you can't really eliminate this stage. Since the "Simulation stage" is going to take in the neighborhood of $\approx f(n)$ steps, this "Compute n stage" will not be excessive provided $f(n) \geq n^2$ (or $f(n) \geq n \log n$ if you're using the clever solution), but it *will* be a bottleneck for smaller $f(n)$ functions.

Remark (Compute $f(|w|)$ stage running time)**.** The "Compute $f(|w|)$ stage" took $O(n \log n)$ time in our case of $f(m) = 2^n$. Notice that this is much much less than the $\approx f(n)$ time for the "Simulation stage", so it's no bottleneck, at least in the case of $f(n) = 2^n$. On the other hand, notice that we needed a whole lemma, Lemma (2^n is time-constructible), to establish that $f(|w|)$ could be computed in $O(n \log n)$ time.

This last remark leads us to a technically-important-but-fairly-annoying point for the study of the Time Hierarchy Theorem. If we want to show a fact like "**BoundedAccepts**$_{f(\bullet)}$ is decidable in not much more than $f(n)$ time" for a general function $f(\bullet)$, we can run into problems *if computing $f(\bullet)$ itself takes an inordinately long amount of time*.

To take an extremely ridiculous example, suppose we decided to care about the following truly insane running time function:

$$f(n) = \begin{cases} n^{\sin(n)} & \text{if the TM } M \text{ whose encoding } \langle M \rangle \text{ is the same as } \langle n \rangle \text{ halts on input } \varepsilon, \\ 2^{2^{2^n}} & \text{if the TM } M \text{ whose encoding } \langle M \rangle \text{ is the same as } \langle n \rangle \text{ loops on input } \varepsilon. \end{cases}$$

Then we would not be able to prove "**BoundedAccepts**$_{f(\bullet)}$ is decidable in not much more than $f(n)$ time", because how would we do the "Compute $f(|w|)$ stage"? (To do this, you'd have to solve the Halting Problem, which can't be done in *any* time bound.)

Luckily, no one in their right mind would ever decide to care about the ridiculous function $f(n)$ above! Instead a normal person is interested in "normal" time functions, like $f(n) = n^2$ or $f(n) = 10n^3$ or $f(n) = 3^n$ or whatever. And it turns out that the analogue of Lemma (2^n is time-constructible) holds for all these "normal" functions: you can compute $f(n)$ in (much) less than $f(n)$ time.

Definition (Clockable functions)**.** Let $f : \mathbb{N} \to \mathbb{R}_{\geq 0}$. We say that f is a *clockable* function if there is a 1-tape TM that computes the function $m \mapsto \lceil f(m) \rceil$ in $O(f(m))$ steps. (Super-technicality: We put the $\lceil$ceiling$\rceil$ in there just because functions like $f(m) = m \log m$ or $f(m) = 1.1^m$ technically don't output integers.) For future simplicity, we will also insist that clockable functions have to be *increasing*, meaning $f(n+1) \geq f(n)$ for all n.

Important. "Clockable" is *455 Slang*; the official formal terms for (variations of) this definition are "time-constructible", "proper complexity", or "honest".

Example. The function $f(m) = 2m^2$ is clockable.

Proof. First, this function is increasing. The main thing we want to show that there is a TM which, given $\langle m \rangle$, outputs $\langle 2m^2 \rangle$ in $O(2m^2) = O(m^2)$ time. Note that the input length is $|\langle m \rangle| \approx \log_2 m$, and the output length is $|\langle 2m^2 \rangle| \approx \log_2(2m^2) = 2\log_2 m + 1$. So we have $O(m^2)$ time to do some relatively basic arithmetic on numbers with $O(\log m)$ digits; in effect, we have *exponential time* (as a function of the input length). This is *way* more time than we need. As you learned in earlier classes, basic arithmetic on ℓ-bit numbers can be done in $\text{poly}(\ell)$ time; we can just do the "grade school multiplication algorithms" in C-like pseudocode, or on a multi-tape Turing Machine, in $O(\ell^2)$ time, and this at worst might translate into $O(\ell^8)$ time on a one-tape Turing Machine. (In fact, it's not hard to show that binary multiplication can be done in $O(\ell^3)$ time on a one-tape Turing Machine; take a look at the example code on Morphett.) So we can produce $\langle f(m) \rangle = \langle 2m^2 \rangle$ from $\langle m \rangle$ in $O((\log m)^8)$ time easily, which is much less than the $O(m^2)$ time we're allowed. □

Fact. *Virtually any "normal"-looking function $f(m)$ with $f(m) \geq \log m$ is clockable. In particular, any function that looks like $f(m) = m^a$ for $a > 0$ a rational number, or like $f(m) = m^a \log^b m$ for $a, b > 0$ rational, or c^m for $c > 1$ rational, or $m!$, or any sum or product of these kinds of functions... they're all clockable. In this course, we promise to never try to "trick" you by talking about non-clockable functions, and you may take for granted that any normal-looking function is clockable unless we direct you otherwise.*

Putting together Remark (Simulation stage running time), Remark (Compute n stage running time), and Remark (Compute $f(|w|)$ stage running time), we conclude:

Theorem (**BoundedAccepts**$_{f(\bullet)}$ time complexity)**.** *Let f be a clockable function. Then there is a universal TM $\mathcal{U}_f$ which, on input $\langle M, w\rangle$ with $n = |\langle M, w\rangle|$, simulates $M(w)$ correctly for up to $f(|w|)$ steps in time*

$$O(n \log n) + O(|\langle M\rangle|^2 + \log f(n)) \cdot f(n) \leq O(n^2 + \log f(n)) \cdot f(n).$$

*For $f(n) \leq 2^{n^2}$, this can be bounded by $O(n^2 \cdot f(n))$; hence for such f we have **BoundedAccepts**$_{f(\bullet)} \in$ $\mathsf{TIME}(n^2 \cdot f(n))$.*

Proof. The $O(n \log n)$ time is for (cleverly) computing $|w|$; but note that this is smaller than the last term $O(n^2 \cdot f(n))$. Also, the time to compute $f(|w|)$ is at most $O(f(n))$ since f is clockable; this is also smaller than the last term $O(n^2 \cdot f(n))$. The $O(|\langle M\rangle|^2 + \log f(n)) \cdot f(n)$ is the time to run the alarm-clocked universal TM, and the inequality is because $|\langle M\rangle| \leq n$. □

Example. For example, if $f(m) = n^3$, then **BoundedAccepts**$_{f(\bullet)} \in \mathsf{TIME}(n^5)$.